Automatic Generation of Scenarios for System-level Simulation-based Verification of Autonomous Driving Systems*

Srajan Goyal
Fondazione Bruno Kessler
University of Trento
Trento, Italy
sgoyal@fbk.eu

Alberto Griggio
Fondazione Bruno Kessler
Trento, Italy
griggio@fbk.eu

Jacob Kimblad
Fondazione Bruno Kessler
Trento, Italy
jkimblad@fbk.eu

Stefano Tonetta
Fondazione Bruno Kessler
Trento, Italy
tonettas@fbk.eu

With increasing complexity of Automated Driving Systems (ADS), ensuring their safety and reliability has become a critical challenge. The Verification and Validation (V&V) of these systems are particularly demanding when AI components are employed to implement perception and/or control functions. In ESA-funded project VIVAS, we developed a generic framework for system-level simulation-based V&V of autonomous systems. The approach is based on a simulation model of the system, an abstract model that describes symbolically the system behavior, and formal methods to generate scenarios and verify the simulation executions. Various coverage criteria can be defined to guide the automated generation of the scenarios.

In this paper, we describe the instantiation of the VIVAS framework for an ADS case study. This is based on the integration of CARLA, a widely-used driving simulator, and its ScenarioRunner tool, which enables the creation of diverse and complex driving scenarios. This is also used in the CARLA Autonomous Driving Challenge to validate different ADS agents for perception and control based on AI, shared by the CARLA community. We describe the development of an abstract ADS model and the formulation of a coverage criterion that focuses on the behaviors of vehicles relative to the vehicle with ADS under verification. Leveraging the VIVAS framework, we generate and execute various driving scenarios, thus testing the capabilities of the AI components. The results show the effectiveness of VIVAS in automatically generating scenarios for system-level simulation-based V&V of an automated driving system using CARLA and ScenarioRunner. Therefore, they highlight the potential of the approach as a powerful tool in the future of ADS V&V methodologies.

1 Introduction

In the rapid evolution of Autonomous Driving Systems (ADS), the problem of ensuring their safety and reliability has become a paramount concern. The Verification and Validation (V&V) of these systems necessitate the assessment of their correctness in a multitude of dynamic and complex real-world scenarios. To address this challenge, the integration of powerful simulation tools with advanced verification methodologies has gained considerable attention [10, 18].

*This work has been supported by: the "AI@TN" project funded by the Autonomous Province of Trento; the PNRR project FAIR - Future AI Research (PE00000013), under the NRRP MUR program funded by the NextGenerationEU; and the PNRR MUR project VITALITY (ECS00000041), Spoke 2 ASTRA - Advanced Space Technologies and Research Alliance.

M. Farrell, M. Luckcuck, M. Schwammberger, and
M. Gleirscher (Eds): Fifth International Workshop on
Formal Methods for Autonomous Systems (FMAS 2023)
EPTCS 395, 2023, pp. 113–129, doi:10.4204/EPTCS.395.8

Under the support of ESA funding, the VIVAS project [12] was dedicated to developing a generic framework tailored for system-level simulation-based V&V of autonomous systems. The approach is based on a simulation model of the system, an abstract model that describes symbolically the system behavior, and formal methods to generate scenarios and verify the simulation executions. It permits the specification of diverse coverage criteria, thereby directing the automated creation of scenarios, and formal properties to be verified on the simulation runs. The framework has been created for space applications and applied to two use cases employing AI for resource prediction and opportunistic science. The system under test is based on the robotic digital twin developed in ROBDT [2].

In the automotive context, the CARLA simulator [9] has established itself as a widely used platform for simulating intricate driving scenarios in a controlled virtual environment. Its ScenarioRunner tool [25] further enhances its capabilities by enabling the specification of diverse and complex scenarios based on the reuse of various predefined car behaviors. Various works proposed AI-based solutions (e.g., [4, 22, 29]) for the perception and control components of cars that are integrated with the CARLA simulator for their validation. The CARLA community also organized a competition to compare and rank such solutions [24]. These autonomous driving agents, grounded in AI methodologies, serve as crucial components in achieving the autonomy of cars. Their integration in CARLA allows evaluating the ADS behavior across different scenarios. However, such validation is so far based on a few manually crafted scenarios.

This paper explores the application of the VIVAS framework to automatically generate scenarios for a system-level simulation-based verification of autonomous driving systems. This integration facilitates a comprehensive assessment of ADS correctness under various conditions, contributing to the enhancement of their safety and reliability. The paper describes the abstract ADS model in the extended SMV language handled by the nuXmv [3] symbolic model checker, capturing essential aspects of its functionality and behavior in a simple highway traffic situation. It then details the formulation of a coverage criterion based on such an abstract model, focusing on the interactions between other vehicles and the vehicle of the ADS under verification (hereafter called *ego*). The VIVAS integration finally consists of a translation of the abstract traces generated from the abstract model to the ScenarioRunner specification and a mapping back of the simulation runs to abstract traces for runtime verification of formal properties. The experimental evaluation shows how VIVAS is able to generate interesting scenarios effectively evaluating the behavior of the AI-based agents.

The rest of the paper is organized as follows: in Section 2, we summarize the related works and compare them with our approach; Section 3 describes in more detail the VIVAS framework and its components; in Section 4, we detail the instantiation to the ADS application; Section 5 shows the results while Section 6 draws conclusions and some directions for future work.

2 Related Work

Over the last decade, we have witnessed significant efforts in the verification of AI-based autonomous systems using formal methods. Many works focus on formal verification of neural networks, for example encoding them into constraint solving (e.g., [14–16]) or using abstraction (e.g., [19, 23]), just to name a few approaches. Our approach instead is rooted in the line of research (e.g., [10, 26, 28]) that tackles the verification at the system level using a simulator. This integrates the AI components, potentially using machine learning (ML) models, for perception or control, in the context of a closed-loop cyber-physical system. As in VerifAI [10], the simulation traces are then formally analyzed with monitoring and runtime verification techniques.

Differently from the mentioned approaches, we exploit an abstract symbolic model to generate automatically the scenarios and define a coverage criterion for the generated test cases. While previous approaches focus on the automated synthesis of the simulation parameters for a specific scenario (e.g., different car movements to change lanes in front of the ego car), we concentrate on the generation of different functional scenarios (e.g., sequences of scenes with different change lanes of non-ego cars). Moreover, in this paper, we map such abstract symbolic scenarios to the scenario specification language of CARLA to verify ADS with different available AI solutions. So, the case study is based on available benchmarks for AI-based ADS taken as is.

There are in fact a variety of scenario specification languages that can be used in this context. VerifAI uses the Scenic language [13, 27] to model the abstract feature space defining the scenarios, which can be instantiated to test cases. Scenic is a probabilistic programming language for scenario generation specifically designed to test the robustness of systems containing AI and ML components by allowing the generation of rare events. It allows the specification of spatial and temporal relationships between objects of a scenario as well as composing several scenarios into more complex ones. By the use of distributions for encoding interesting parameters, Scenic will perform automatic test case generation through the use of sampling. Similarly, the Paracosm [18] framework is a programmatic interface that can be used to create various automotive driving simulation scenarios through the design of parameterized environments and test cases. The parameters control the environment in the scenario including the behavior of the actors and can include things such as pedestrians, lanes, and light conditions. Parameters are specified using either discrete or continuous domains and test cases are instantiated from the domain using random sampling and Halton sampling respectively. A coverage criterion is then defined over the coverage of the domains, where k-wise combinatorial coverage is used for the discrete domains and dispersion is used for continuous domains. Although Paracosm can provide output using the OpenDRIVE format, it is primarily coupled to be used with the Unity game engine, and as such scenarios are modeled using the C# programming language. The Measurable Scenario Description Language (M-SDL) [11] is another scenario description language similar to Scenic. In M-SDL, one captures the behavior of identified actors in scenarios. M-SDL makes use of pre-defined basic building blocks such as actors (including the AV) along with some pre-defined behaviors, sets of possible routes, and environmental conditions. Libraries then use the basic building blocks to implement more complex behaviors such as cars overtaking, running red lights, driving on a highway, etc. Since M-SDL scenarios are abstract and parameterized, a single scenario can map onto many concrete ones through the use of sampling. ScenarioRunner [25] is a module of CARLA that allows traffic scenario definition and execution for the CARLA simulator. The scenarios can be defined through a Python interface or using the OpenSCENARIO standard [1]. ScenarioRunner is used to validate AI solutions for ADS. These results can be validated and shared in the CARLA Autonomous Driving Leaderboard [24], an open platform for the community to fairly compare their progress, evaluating agents in realistic traffic situations.

For all the above languages, the scenario must be specified manually, to then derive the test cases automatically. VIVAS instead provides a model-based approach to generate the scenarios automatically based on a coverage criterion that defines the interesting combinations of situations. In this paper, we focus on the integration with CARLA, because it allows the verification of the solutions shared by the CARLA community. However, the approach can also work with different specification languages, and we have, for example, a prototype integration with Scenic interfaced with CARLA. We have not presented the results of this integration in this paper since the ego model is based on Newtonian physics, with no AI models involved in the autonomous driving pipeline.

Although not specifically focused on AI-based systems, another very relevant work is described in [17], which proposes an optimization-based approach to synthesize ADS scenarios from formal spec-

ifications and a given map. Their formal specification of scenarios corresponds to our abstract scenario and is also synthesized from a symbolic model. However, test case generation does not follow any coverage criteria but enumerates specifications starting from an initial scene. In principle, our coverage-driven generation of scenarios can be combined with various techniques to concretize the scenario with different trajectories and sampling of the different environment parameters.

TAF [20] is another tool for automated test case generation of autonomous systems. Their abstract model is defined in an XML-based domain-specific language. It includes semantic constraints on the initial conditions of the environment and its agents (unlike the additional state transition systems in our work), which are solved using SMT solvers to generate abstract test cases. Random sampling is combined with these solvers to diversify the test cases, with an expert given coverage of data values. Their coverage criteria is based on covering parameter values to instantiate the scenarios. Although constraints on time can be expressed, more generic temporal specification on the sequences of actions and the related coverage criteria are not supported as in our approach. On the other side, our framework can be extended to constraints with quantifiers and complex data structure as in [21], which are currently not supported in VIVAS.

3 The VIVAS Framework

VIVAS is a V&V framework for generating test cases for autonomous systems (possibly using AI/ML components) via a combination of system-level simulation and symbolic model checking. VIVAS makes use of formal, symbolic models of the environment and system components to generate *abstract test scenarios* for the autonomous system of interest using model checking techniques. The abstract test scenarios are then instantiated by the concretization of the abstract parameters to provide concrete scenarios to be executed on a system-level simulator encompassing AI/ML models, to obtain execution traces that are in turn analyzed by an automatically-generated monitor. The output of the framework is a V&V result consisting of coverage statistics of the executed traces with respect to the symbolic models and quantitative and qualitative information for each use case. The overview of the architecture can be seen in Fig. 1, which depicts the main parts of the VIVAS framework. These are the abstract scenario generator, the concrete scenario generator, the simulator, and the executor monitor.

Abstract Scenario Generation. Scenario generation is the first step of the approach. The starting point is a formal, symbolic model of the system, which provides an abstract view of both the environment and the components under test (including AI/ML parts). ML components are defined in a declarative manner, approximated in terms of input-output mapping. Abstract test scenarios are generated from the formal system model using symbolic model checking techniques by the abstract scenario generator. Abstract scenarios are defined as combinations of values of predicates describing interesting behaviors of the abstract system. From the technical point of view, each abstract scenario is encoded as a formal property that is expected to be violated by the system (i.e. a property specifying that "the scenario cannot occur in the abstract system"). For each such property defined by the abstract scenario generator, a model checker will be executed on the system model, with the goal of finding a counterexample to the property. By construction, each such counterexample corresponds to an execution trace witnessing the realization of the abstract scenario of interest.

Concrete Scenario Generation. Each of the traces produced by the model checker is then refined into a (set of) concrete scenarios that can be used to drive the system-level (concrete) simulator. Due

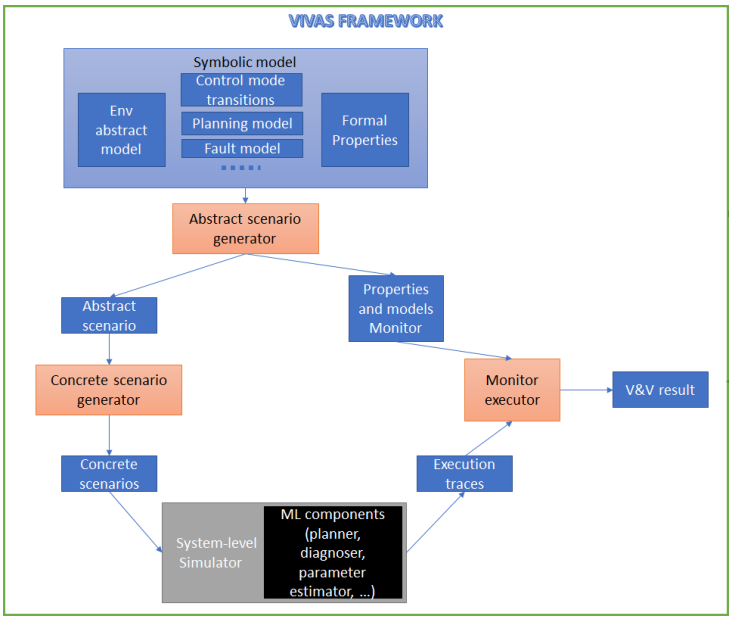

Figure 1: Top-level architecture of VIVAS framework. Blue boxes: artifacts, Orange boxes: code.

to uncertainties and abstractions in the abstract scenarios, a one-to-many mapping is defined where a single abstract scenario can be instantiated to many, possibly infinite, concrete scenarios. This is done by defining a mapping between the abstract values and the set of concrete values that they represent and then appropriately sampling from the sets. An example is that the abstract model might discretize the time of day into dusk, dawn, midnight, and midday. The concretizer then has to sample the actual time for the simulation.

Simulation. The task of the system-level simulator is to run a simulation of the target asset under the requested conditions, by configuring the system, its environment, and its inputs as specified in the concrete scenario produced by VIVAS. Upon completion, the simulator provides the corresponding execution trace of the system, containing all the necessary details to evaluate the properties of interest.

Execution Monitor. Each concrete scenario produced is executed by the simulator, which generates a corresponding concrete execution trace. This trace is then used to determine whether:

1. the concrete execution of the system satisfies the property of interest, and
2. the concrete execution of the system complies with the input abstract scenario (which defines the situation of interest for the current test).

This is done by formally evaluating the trace with a runtime monitor that is automatically generated from the formal specification of the property and the abstract system model. The trace evaluation can have four possible outcomes:

1. The trace complies with the abstract scenario (defining the situation under test), and it also satisfies the property: the test execution is relevant and the test passes.
2. The trace complies with the abstract scenario, but it does not satisfy the property: this corresponds to a test failure on a relevant scenario, and it should be reported to the user.

3. The trace satisfies the property, but it does not comply with the abstract scenario: this corresponds to a (good) execution in an unexpected situation, in which some of the assumptions defining the scenario might be violated. This might be due to imprecisions/abstractions in the symbolic model and in the concretizer, which might prevent the realization of the abstract scenario under analysis. This situation might be reported to the user, as it might suggest that a revision/refinement of the symbolic model might be needed.
4. The trace violates the property and it does not comply with the abstract scenario: this corresponds to a test failure in an unexpected situation. Similarly to the above, it might be a warning that the symbolic model of the system is not precise enough to capture the situations of interest defined by the abstract scenario.

Abstract and concrete coverage. Ensuring an adequate level of coverage is one of the primary goals of a good set of tests. In VIVAS, coverage is defined with respect to a domain-specific notion of "interesting situations", which are those that are (implicitly) defined by the possible combinations of values of predicates that are used by the abstract scenario generator to produce abstract traces. By construction, therefore, VIVAS tries to enumerate abstract scenarios that ensure a 100% degree of coverage of the *abstract* situations of interest[1]. Each abstract scenario is then refined into one or more concrete simulation inputs, leading to corresponding concrete simulation traces. In order to determine the *concrete* coverage (i.e., the degree of coverage of interesting situations at the concrete level), the VIVAS monitor analyzes the execution traces. It checks for compliance with the property of interest and the corresponding abstract scenario's specification (i.e., the "interesting situation") from which the concrete executions originate.

Only executions that satisfy the abstract scenario specification contribute to the coverage at the concrete level: if an execution does not comply with its abstract specification, it represents an unexpected situation from which no coverage information can be drawn[2].

4 Autonomous Driving Application

In order to apply the VIVAS methodology to ADS application, we instantiate various components of the VIVAS framework. We choose CARLA simulator as it is widely used in the automotive domain and it has a large community that provides various AI-based solutions for perception and control. We define an abstract model that focuses on highway scenarios where the ego is surrounded by other vehicles in various dynamic situations. In the following, we provide details about the different components.

4.1 CARLA Simulation Model and AI Components

CARLA [9] is a high-fidelity open-source simulator that provides a dynamic environment for the development, testing, and validation of AD systems. It is written in C++ as a plugin for Unreal Engine. As a standalone package, it provides pre-defined maps with 3D meshes ranging from city roads with intersections to highways, to mimic real-world landscapes for the agents to drive in. Various sensor models

[1]Note that a 100% degree of coverage might not be reached, either because some situations are not feasible already at the abstract level, or because the model checker cannot find a witness trace for the scenario specification within the given resource budget (time and/or memory).

[2]Note that in principle such a situation might still provide *some* information (e.g. it might still cover a different but still interesting situation); therefore, the test result is still reported to the user. However, determining this might not be obvious in general, and therefore we opted for the conservative choice of excluding the test from the computation of the degree of coverage in such cases.

(cameras, Lidar, radar, GPS, IMU) are provided to gather the data from the environment. The simulator includes many vehicle models, from small cars to large trucks, with different properties like mass, dynamics, and controls. A simulation is composed of (i) the CARLA Simulator that computes the physics and renders the scene and all actor properties, (ii) client scripts written using a Python API, that allows control of the actors, sensors, and environmental conditions.

AI-based components. The CARLA community through its leaderboard competition [24] provides various state-of-the-art AI solutions for end-to-end autonomous driving. However, only a few of them provide the necessary code and well-trained models for their methodologies to be evaluated and built upon. We specifically tested Interfuser [22], TCP [29], and LAV [4], all three currently in the top 5 of the leaderboard. Within a few test runs of the AI agent provided with TCP, we noticed that the ego vehicle brakes to a standstill as soon as any other vehicle arrives next to it in its adjacent lane. We consider it too conservative of an autonomous behavior to test our verification methodology. The LAV agent on the other hand behaved well autonomously (in accordance with its overall score on the leaderboard) in terms of route completion and collision avoidance. However, it had an erratic behavior of changing lanes non-deterministically at scenario instantiation. It would require us to make ad-hoc changes to relative positions of the non-egos with respect to ego in every concrete scenario we generate.

We therefore chose the Interfuser agent as the AI system under test for our V&V methodology. It is currently ranked 2 on the leaderboard (rank 1 among the open-source solutions). This solution primarily focuses on the safety of AD systems by generating interpretable semantic features of the environment through multi-model sensor fusion, for constraining the agent's low-level control actions in real-time within safe sets. The perception system processes the data gathered by 3 RGB cameras and one Lidar sensor.

All three AI agents mentioned above share the following main characteristics:

- The maximum driving speed is limited to 5 m/s, which is quite conservative for highway driving;
- The ego always travels in its own lane: an external route for the ego to follow needs to be provided. It may change lanes only based on the waypoints of this route on the map. Hence, it never overtakes slow-moving cars in front of it in the same lane. Ego just follows them while maintaining a safe distance, or keeping a stand-still.
- Standard rules of the road for overtaking only on the left (or the right) are not applied.

Note that our V&V methodology is agnostic to the AI solution chosen for the simulator. Since the abstract test scenarios are generated from the symbolic model of the system, abstract coverage would be the same for different AI solutions, although the concrete coverage may vary. In future work, we will use our methodology to benchmark other AI solutions as well.

4.2 Abstract Model and Coverage Criterion

We specify our abstract model as a synchronous symbolic transition system written in the language of the nuXmv [3] model checker. The model consists of 3 vehicles (one "ego" car, representing the autonomous system under test, and two other cars) moving on a highway with 3 lanes. The vehicles all drive in the same direction. The ego is constrained to stay in the middle lane and tries to maintain a given cruise speed, braking when necessary to avoid collisions with other cars, and possibly accelerating to reach the target speed. The other two "non-ego" cars can move freely on the highway, with arbitrary accelerations, braking, and lane change maneuvers (subject to physical constraints about min/max acceleration rates

```
MODULE Car(id)
IVAR acceleration : real;
VAR pos : real;
    lane : 0 .. MAX_LANE;
    speed : real;
DEFINE changing_lane := next(lane) != lane;
TRANS
  changing_lane -> (speed <= max_lane_change_speed &
                    next(speed) <= max_lane_change_speed);
TRANS
  changing_lane -> (acceleration <= max_lane_change_acceleration &
                    acceleration >= (- max_lane_change_braking));
TRANS
  next(speed) = max(speed + acceleration * TIME_STEP, 0);
TRANS
  changing_lane ?
     (next(pos) = pos + (speed + next(speed)) / 2 * TIME_STEP * 0.95) :
     (next(pos) = pos + (speed + next(speed)) / 2 * TIME_STEP);
TRANS
  (next(lane) = lane) | (next(lane) = lane + 1) | (next(lane) = lane - 1);


MODULE Ego(car1, car2)
-- VAR declarations...
DEFINE
  time_to_stop := speed / (-MAX_BRAKING);
  collision_next := (car1.lane = lane & car1.pos >= pos & speed > 0 &
                       (car1.pos - pos) / speed <= time_to_stop) |
                    (car2.lane = lane & car2.pos >= pos & speed > 0 &
                       (car2.pos - pos) / speed <= time_to_stop);
TRANS
  collision_next ?
  (acceleration = MAX_BRAKING & target_speed = 0) :
  (target_speed = EGO_CRUISE_SPEED &
  ((speed < target_speed) ->
                (next(speed) = min(target_speed,
                                   speed + MAX_ACCELERATION * TIME_STEP))));

INVAR -- the cars do not crash into each other on purpose
  ((abs(pos - car1.pos) > SAFE_DISTANCE) | (lane != car1.lane)) &
  ((abs(pos - car2.pos) > SAFE_DISTANCE) | (lane != car2.lane)) &
  ((abs(car1.pos - car2.pos) > SAFE_DISTANCE) | (car1.lane != car2.lane));
```

Figure 2: Excerpt of the nuXmv code for the abstract model.

and speed limits, taken from publicly available online car databases), but are not allowed to crash into each other or the ego. We use a discrete model of time, in which each transition of the system corresponds to a time-lapse of 1 second. We use the theory of real arithmetic to encode the transition relation of the system, using mostly linear constraints to compute the updates to the speed and locations of the vehicles (thanks to the discretization of time). An excerpt of the symbolic model is shown in Fig. 2. The module `Car` is shared by different non-ego vehicles. Different transition relations on speed, acceleration, and position need to hold when a non-ego changes lane (with `changing_lane`). For the `Ego` module, we define the collision condition (`collision_next`) with non-ego vehicles (`car1` & `car2`, in this case). If True, the ego brakes with the `max_braking` until it stops; else it continues with (or reach towards) its `target_speed`.

In order to enumerate abstract scenarios encoding potentially-interesting traffic situations, we define for each non-ego car a set of predicates specifying its position relative to the ego, in terms of occupation

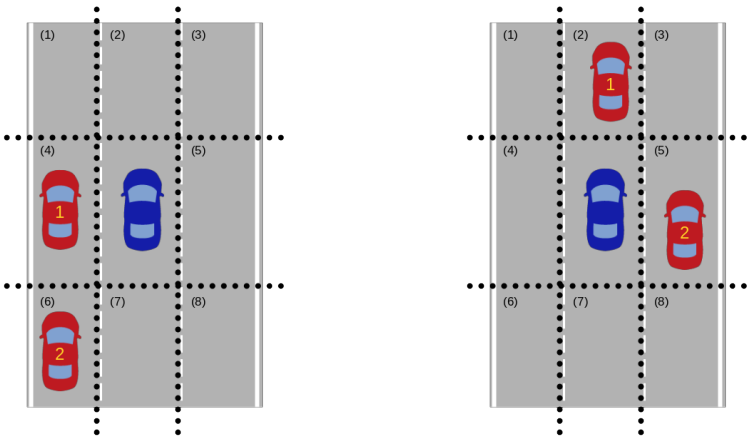

Figure 3: Example traffic situations for constructing abstract scenarios, specifying positions of non-ego cars (in red) in terms of the occupation of cells of an abstract 3x3 grid centered on the ego car (in blue).

of cells of an abstract 3x3 "grid" centered on the ego. Examples of the possible configurations that can be expressed in this way are shown in Fig. 3. We then define an *abstract scenario* as a combination of constraints about the different positions of the non-ego cars on the grid at different points in time. More specifically, each abstract scenario is specified as an LTL property of the following form:

$$\begin{aligned}\neg\mathbf{F}(&\text{car1\_grid\_pos} = \text{CELL\_A1} \wedge \text{car2\_grid\_pos} = \text{CELL\_A2} \wedge \\ &\mathbf{X}(\mathbf{F}(\text{car1\_grid\_pos} = \text{CELL\_B1} \wedge \text{car2\_grid\_pos} = \text{CELL\_B2}))),\end{aligned} \tag{1}$$

(where car$i$_grid_pos encodes the position of the i-th non-ego car in the grid and CELL_$*$ represent possible target positions for the cars.) By asking the model checker to find a counterexample to Eq. 1, we generate traces in which the non-ego cars first reach the configuration with `car1` in position A1 and `car2` in position A2, and then subsequently move to the configuration with `car1` in position B1 and `car2` in position B2, performing the necessary maneuvers while avoiding collisions with each other or with the ego.

The space of scenarios that is being explored therefore consists of all the possible combinations of transitions from configurations of the non-ego cars in terms of their position in the grid defined above. Enumerating all of them would give 4096 scenarios. We define our coverage criterion by selecting a subset of *abstract scenarios of interest*, consisting of various combinations of the traffic situations that can be modeled by positioning the non-ego cars in the grid around the ego. In total, for the experiments, we defined 144 such interesting scenarios.

4.3 VIVAS Interface with CARLA

In order to generate a scenario for the CARLA simulator, the abstract counterexample trace generated by the model checker is parsed for relevant information to be fed as input to the simulator. As an interface to the simulator, we used the CARLA module ScenarioRunner. This provides a Python interface to specify the routes for the ego as well as complex traffic scenarios by defining the behavior of the non-ego agent(s). ScenarioRunner also allows for running CARLA on a specified map at a particular location, while the user is allowed to implement their own AI-based ego agent. Every state of the abstract scenario trace is concretized into the corresponding behavior of every non-ego agent. Each behavior is then specified in

```
1  # instantiate car1's sequential behaviors
2  car1_behavior = py_trees.composites.Sequence("car1_behavior")
3
4  # first behavior in the sequence for car1: to drive forward.
5  ##   Atomic behavior WaypointFollower runs in parallel with an atomic trigger condition DriveDistance
6  drive_car1 = py_trees.composites.Parallel("Drive forward",
7                  policy=py_trees.common.ParallelPolicy.SUCCESS_ON_ONE)
8  drive_car1.add_child(WaypointFollower(car1, speed=3, avoid_collision=False))
9  ## drive_car1 terminates when car1 reaches the given distance
10 drive_car1.add_child(DriveDistance(car1, distance=2.6))
11 car1_behavior.add_child(drive_car1)
12
13 # second behavior in the sequence for car1: to change lane
14 lane_change_car1 = py_trees.composites.Parallel("lane change",
15                   policy=py_trees.common.ParallelPolicy.SUCCESS_ON_ONE)
16 ## change lanes to the left with a speed of 2 m/s
17 lane_change_car1.add_child(LaneChange(car1, speed=2, direction='left',
18             distance_same_lane=1.3, distance_other_lane=60, distance_lane_change=9))
19 lane_change_car1.add_child(DriveDistance(car1, distance=12))
20 car1_behavior.add_child(lane_change_car1)
21
22 # third behavior in the sequence for car1: to stand still for 1 second.
23 still_car1 = py_trees.composites.Parallel("standstill",
24                 policy=py_trees.common.ParallelPolicy.SUCCESS_ON_ONE)
25 still_car1.add_child(WaypointFollower(car1, speed=0.0, avoid_collision=False))
26 still_car1.add_child(StandStill(car1, name="StandStill", duration=1.0))
27 car1_behavior.add_child(still_car1)
```

Figure 4: Excerpt of automatically generated Scenario runner code (in Python3) for a non-ego (car1) behavior. Comments (in green) explain the behavior tree.

Python to generate a behavior tree for each corresponding non-ego vehicle. The behavior trees of all the non-egos present in the environment are then run in parallel during the simulation.

We first parse the initial coordinates and lanes of all the non-egos relative to the ego to instantiate them on the map. In each behavior of a behavior tree, the corresponding non-ego has to drive at a certain speed for a certain distance, following the waypoints given by the map on the same lane it is instantiated on. Although the duration of each state transition in the abstract trace is 1 second, the non-egos may take longer to drive that particular distance in the CARLA simulator, due to potential mismatches between the symbolic model and the simulator models. In case the vehicle stands still for n states in the abstract trace, it stands still for n seconds in the concrete scenario once it comes to a halt.

In the symbolic model, lane changes occur in one time step, with zero lateral distance traveled (since lanes have no width in the symbolic world). However, we constrain the successive lane changes of the same car to be N steps apart[3] to model the fact that a lane change is not instantaneous overall. To concretize this particular state transition, the non-ego transverses $9m$ while changing lanes, with this behavior terminating after traveling a total distance of $12m$ for the next behavior in the tree to be instantiated. Below these values, lane changes were not possible in CARLA at the speed ranges the vehicles drive in our scenario. Note that similar to the symbolic model, a non-ego can change only one lane at a time, with inputs $\{\text{left}, \text{right}\}$ meaning change lane to the left or to the right.

We leverage the behavior library of ScenarioRunner to write these atomic behaviors and trigger conditions. An example behavior tree for one non-ego (car1) with all the above explained three behaviors is shown in Fig. 4. Here, lines 6-11: drive straight forward for 2.6 m, with a speed of 3 m/s; lines 14-20: perform a lane change to the left, with a speed of 2 m/s, driving a total of 12 m within which 9 m is the distance traveled while changing lanes; lines 23-37: stand still for 1 second. We do not need to

[3]We used $N = 6$ in our experiments.

extract ego's behavior from the abstract trace, since it is expected to make decisions autonomously in the simulator. Only initial spawn position and destination need to be extracted for the AI-based agent to follow the route.

The concrete simulation traces are then mapped back to the abstract trace to measure coverage, to check if the same sequence of scenes was encountered in the concrete scenarios or not. In particular, a predicate map is defined to map the absolute positions of the non-egos in the concrete simulation trace to the abstract 3x3 grid shown in Fig. 3. Here, we show examples of mapping the positions of non-egos to the cell locations 1,4 and 8 of the abstract grid:

$$\begin{aligned}
&\textbf{CELL_1} : (\text{car_i.lane} < \text{ego.lane}) \wedge (4 \leq |\text{car_i.pos - ego.pos}| \leq 24) \wedge (\text{car_i.pos} > \text{ego.pos}) \\
&\textbf{CELL_4} : (\text{car_i.lane} < \text{ego.lane}) \wedge (|\text{car_i.pos} - \text{ego.pos}| \leq 10) \\
&\textbf{CELL_8} : (\text{car_i.lane} > \text{ego.lane}) \wedge (4 \leq |\text{car_i.pos} - \text{ego.pos}| \leq 24) \wedge (\text{car_i.pos} < \text{ego.pos}),
\end{aligned} \tag{2}$$

where car_i.pos is the longitudinal position (in meters) of the i-th car in the simulation trace (and similarly for ego.pos). In this way, we define the boundaries of the cells on the abstract grid.

4.4 Monitoring of Properties

As described above, the monitor component of VIVAS is used to determine whether the concrete system (simulator) satisfies the system-level formal specification. For the automotive application, the simulation output traces include sequences of all states and actions executed by the ego vehicle, along with the time evolution of other observable parameters, which must be checked for property satisfaction/violation. In this study, we primarily need to check whether the ego vehicle crashes with another vehicle in the environment. Since the ego always travels in its own lane, we limit the check for the case when the ego crashes with any non-ego in front of it in its own lane. We do this by leveraging the continuous data stream from the collision sensor mounted on the ego. The monitor is currently hard-coded for monitoring specifically the output of this sensor, i.e., it checks whether the ego crashes or not at any time step in the simulation trace. Along with the satisfaction/violation of this property, the positions of non-egos in the simulation trace are mapped back to the abstract grid, to measure the degree of concrete coverage as described in §3.

In the future, we plan to use a runtime monitor based on NuRV [7, 8], to check standard LTL properties on ego behavior, e.g., if ego brakes within n time-steps as soon as any non-ego comes in front within its safe driving distance, or if the lane change of another vehicle is detected by the perception component of the ego within m time steps.

5 Results

In this section, we report on our experimental evaluation of our instantiation of VIVAS for the ADS application using the CARLA simulator. We first describe the experimental setup in §5.1, including the choice of parameters for the vehicles and environment in the symbolic model and in the CARLA simulator, necessary to generate meaningful scenarios. We then present the results of the evaluation in §5.2 and discuss them in §5.3.

We ran the experiments on an Intel i7 with NVIDIA GeForce RTX 2080 8GB GPU. These are the minimum hardware requirements to run the simulations on the CARLA simulator with AI models. All the experiments take roughly 22 hours to complete. We used a timeout of 200 seconds for each abstract scenario generation. This timeout was never reached by the model checker during our experiments: on

average, model checking took less than 10 seconds for each instance. Rather, the performance bottleneck turned out to be the time to instantiate a scenario in CARLA and perform the simulations, which took 3 minutes on average.

The code and data necessary for reproducing our experiments are available at `https://es-static.fbk.eu/people/sgoyal/fmas23`.

5.1 Experimental setup

At the level of the symbolic model, we define here some fixed parameters for simple, but meaningful scenario generation:

$$\begin{array}{lll} \text{ego\_cruise\_speed} & = 5 & (m/s) \\ \text{non\_ego\_speed} & = [0,\ 12] & (m/s) \\ \text{max\_acceleration} & = 5.6 & (m/s^2) \\ \text{max\_braking} & = -4.6 & (m/s^2) \\ \text{safe\_distance} & = 7 & (m) \\ \text{lanes} & = \{\text{left},\ \text{center},\ \text{right}\} & \end{array} \tag{3}$$

All the scenarios that we generate consist of 2 non-ego agents and one ego agent, all of which start from the same longitudinal position $x = 0$, with ego in the center lane and 2 non-egos on each side of it. Note that fixing the initial positions would not make a difference to the abstract scenario generation since the acceleration, braking, and speed for the non-egos are picked non-deterministically by the model checker for every time step, while respecting the above bounds. Since the AI-based agent we use in CARLA can only drive at around 5 m/s, we limit the ego agent cruising speed to the same. All the agents start from 0 m/s, with ego reaching its cruising speed with max_acceleration. To avoid collisions, it brakes with max_braking to maintain at minimum the safe_distance with all the non-egos.

To improve the robustness of abstract scenario generation, we reduce the size of each cell in the abstract 3x3 grid in the symbolic model (see Fig. 3) by 3m in each direction, compared to the grid we use for evaluation of the coverage on the simulator. The lower values of the relative distances here are chosen to specifically create situations where non-egos stay close to ego and challenge its perception and control system with their braking and lane-changing maneuvers.

To mimic the symbolic environment model, we instantiate the CARLA simulator on a section of a highway of Town06, with 5 straight lanes, with ego positioned on the center lane and two non-egos on each lane next to it, corresponding to the symbolic model. Left-most and right-most lanes are not used. To compensate for the mismatch between the vehicle dynamics in the symbolic model and CARLA simulator, we concretize the initial positions of the non-egos at:

$$x = \{-3.5, 0, 3.5\}\ m \tag{4}$$

i.e., the non-egos start at 3.5 m behind, same level, and 3.5 m ahead of the ego in their respective lanes in different simulation runs. All the vehicles start from 0 m/s, as parsed from the abstract trace. In future work, we could also concretize further for the simulations with one non-ego ahead and the other one behind the ego, to check if it extends the coverage results.

CARLA provides the possibility to change weather conditions (e.g., rainy, cloudy, night, etc.) at the beginning of simulation runs. We perform all the simulations in "clear noon" setting, for the ego's perception components to operate in the least challenging conditions. In future work, we will evaluate the AI solution in different weather conditions.

Table 1: Coverage Results.

Non-ego position	**Total Scenarios**	**Coverage OK**	**Property FAIL**	**Coverage OK ∩ Property FAIL**
3.5 m	144	68	3	2
0	144	73	17	4
- 3.5 m	144	45	54	19
Set Union		**81**	**61**	**25**

5.2 Evaluation

The model checker produces a total of 144 abstract scenarios based on the coverage criteria given in §4.2. Each abstract scenario is concretized into three concrete scenarios, by varying the initial positions of the non-egos according to Eq. 4, which gives us $144 * 3 = 432$ concrete scenario outputs from the simulator. The evaluation results are shown in Table 1. The columns have the following meanings:

Coverage OK: Each point of the grid of coverage criteria represents a scenario with a fixed order of scenes. The concrete simulation run passes ("OK") if the abstract scenario generated by the model checker could indeed be generated on the simulator as well.

Property FAIL: The system-level property fails if the autonomous ego agent collides with at least one non-ego in front. We do not take into account the situations where non-egos crash into each other or hit the ego from behind.

Coverage OK ∩ Property FAIL: the intersection of the above two conditions. These are the set of "interesting" cases (along with the other cases where property failed), where the coverage criteria passed, but the ego crashed with a non-ego in front.

Set Union: Combines the results for all concrete scenarios with respect to the abstract scenarios.

5.3 Analysis

All intended 144 abstract counterexamples could be generated by the model checker, meaning that the configurations we defined for the coverage did not violate any constraints in the symbolic model. As we see from the obtained results, not all the abstract scenarios generated by the model checker could be covered in the simulator. We could cover a total of only 81 out of 144 scenarios. This is primarily due to the mismatch in (a) behavior models and (b) vehicle dynamics, between the symbolic model and the simulator.

Behavior model mismatch. Since the ego is based on AI models, its non-deterministic behavior is not fully represented in the symbolic model. Ego speed is always varying within +-1 m/s compared to the constant cruising speed in the symbolic model. The bounds of cells in the predicate map are defined for the relative position of the non-egos with respect to ego, as specified in Eq. 2. Hence, in some cases, the non-egos can not reach the required region within these bounds during the scenario, since the ego is traveling too fast or slow. In principle, we could overcome this by conditioning the non-egos' behavior to the ego's in terms of the distance traveled relative to the ego instead of the absolute distance on the lane, while translating the abstract scenario to the concrete one. However, no such atomic behaviors or conditions yet exist in ScenarioRunner. This could be included in one of the future works.

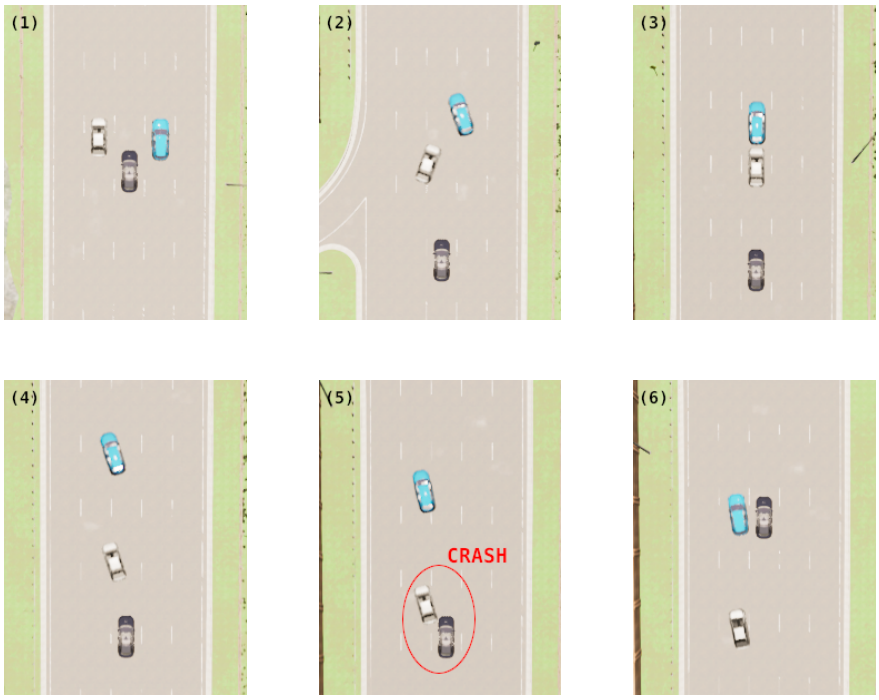

Figure 5: Scenario defined in Eq. 5, with Coverage OK ∩ Property FAIL. Ego crashing with a non-ego in front (scene 5).

Vehicle dynamics mismatch. The vehicle dynamics models are based on OEM data, hard-coded in the simulator. The physical constraints for maximum acceleration and braking in the symbolic model are instead generic. However, we found a big mismatch during the simulations, with maximum acceleration values of vehicle models in CARLA reaching as high as 12 m/s^2, and maximum braking below -15 m/s^2. Since these parameter values also vary from vehicle to vehicle in CARLA, we even came across situations where non-egos crashed into each other while changing lanes.

Interesting scenarios. Even though we could not cover all the abstract scenarios in the concrete simulator, there were 61 scenarios where AI-based ego collided with a non-ego agent in front. In particular, we obtained 25 interesting scenarios that met their abstract specification, but with ego crashing into a non-ego in front. Fig. 5 shows 6 scenes (in temporal order of 1-6) extracted from one such scenario. This corresponds to the LTL property specified below in Eq. 5 (with reference to Eq. 1):

$$\neg\mathbf{F}(\text{car1\_grid\_pos} = 2 \wedge \text{car2\_grid\_pos} = 2 \wedge \mathbf{X}(\mathbf{F}(\text{car1\_grid\_pos} = 6 \wedge \text{car2\_grid\_pos} = 4))) \quad (5)$$

Here, the grid positions correspond to the cell numbers mentioned in Fig. 3 for the abstract grid space. We now describe the scenario in Fig. 5.

- All the agents are initialized at 0 m/s, with car1 on the left of ego, and car2 to its right, with both cars starting 3.5 m ahead of ego (scene 1). Here, "car1_grid_pos = 4 ∧ car2_grid_pos = 5" on the abstract grid.
- The non-egos travel faster than the ego to change lanes in front of it (scene 2), and end up in the configuration (scene 3) where the first predicate, "car1_grid_pos = 2 ∧ car2_grid_pos = 2" is satisfied.

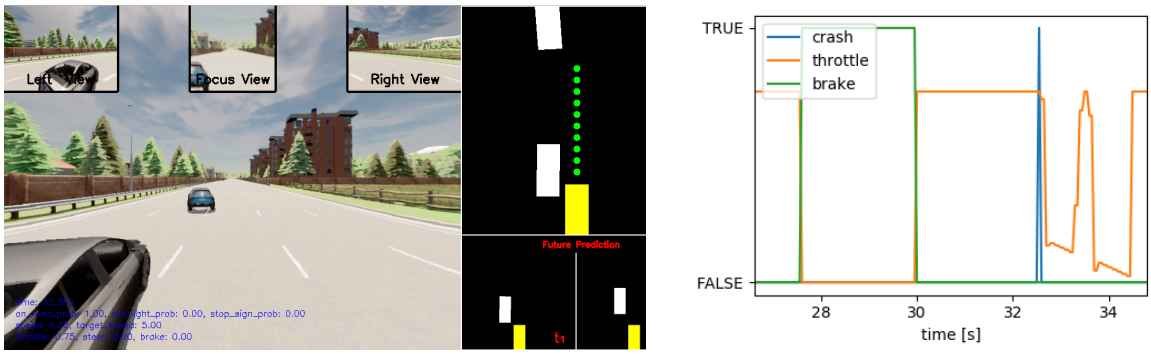

Figure 6: Ego collision; (left): Different camera views of AI-based ego, with the perception component's output; (right): Plot showing ego's throttle and braking values during the collision (in blue).

- The non-egos then change lanes to the left slowly (scene 4), when the ego crashes with car1 (scene 5), even when the car1 has still not completed the lane change.

  Fig. 6 (left) shows the front view of the AI agent at the instant of crashing with non-ego. The real-time telemetry shows that ego's brake = 0 and throttle = 0.75. The perception component here seems to mis-detect the actual position of car1. The corresponding output from the simulation trace is shown in the right plot, with ego's throttle = True (and brake = False) for more than 2 seconds leading to the collision.

- The ego keeps driving forward when the second predicate, "car1_grid_pos = 6 ∧ car2_grid_pos = 4" is satisfied (scene 6).

6 Conclusions

This paper showed the application of system-level simulation-based verification of ADS using formal methods to generate abstract scenarios. The verification toolchain includes nuXmv for model checking and generating abstract scenarios, CARLA for simulating the ego behaviors in concrete scenarios, and mappings from abstract to concrete scenarios and back. We presented an abstract model of the system and a coverage criterion that allows the automated generation of abstract scenarios with model checking. The generated abstract scenarios cover different sequences of traffic scenes that are relevant to test the reaction of the ego's behavior to see if it avoids crashing into other cars. The simulation with CARLA of the corresponding concrete scenarios showed various crashes caused by the ego, although not all simulations reproduce the expected abstract scenario. Inspecting some of the simulations reporting a crash in a covered abstract scenario confirms that the ego behavior is indeed buggy and this is probably due to the AI-based perception component.

During this study, we gained many insights that may lead to some interesting future research directions. These include more efficient techniques to generate abstract scenarios for minimizing the number of model checking runs needed to achieve a certain coverage level; the integration of effective sampling techniques that synthesize various simulation parameters for the same abstract scenario; extending the abstract model by incorporating uncertainty in the ego behavior or a more precise representation of the continuous-time behavior with timed or hybrid version of SMV [5, 6]; finally, enhancing the concrete scenario specification with conditional behaviors of non-ego vehicles that react to the choices of the ego.

References

[1] Associaiton for Standardization of Automation & Measuring Sytems (ASAM): *OpenSCENARIO*. `https://www.asam.net/standards/detail/openscenario/`. Accessed: 2023-08-30.

[2] Marco Bozzano, Riccardo Bussola, Marco Cristoforetti, Srajan Goyal, Martin Jonáš, Konstantinos Kapellos, Andrea Micheli, Davide Soldà, Stefano Tonetta, Christos Tranoris & Alessandro Valentini (2023): *RobDT: AI-enhanced Digital Twin for Space Exploration Robotic Assets*. In: *The Use of Artificial Intelligence for Space Applications*, Springer Nature Switzerland, pp. 183–198, doi:10.1007/978-3-031-25755-1_12.

[3] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri & Stefano Tonetta (2014): *The nuXmv Symbolic Model Checker*. In: *CAV*, *Lecture Notes in Computer Science* 8559, Springer, pp. 334–342, doi:10.1007/978-3-319-08867-9_22.

[4] D. Chen & P. Krahenbuhl (2022): *Learning from All Vehicles*. In: *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE Computer Society, Los Alamitos, CA, USA, pp. 17201–17210, doi:10.1109/CVPR52688.2022.01671.

[5] Alessandro Cimatti, Alberto Griggio, Enrico Magnago, Marco Roveri & Stefano Tonetta (2019): *Extending nuXmv with Timed Transition Systems and Timed Temporal Properties*. In Isil Dillig & Serdar Tasiran, editors: *Computer Aided Verification*, Springer International Publishing, Cham, pp. 376–386, doi:10.1007/978-3-030-25540-4_21.

[6] Alessandro Cimatti, Alberto Griggio, Sergio Mover & Stefano Tonetta (2015): *HyComp: An SMT-Based Model Checker for Hybrid Systems*. In: *TACAS*, *Lecture Notes in Computer Science* 9035, Springer, pp. 52–67, doi:10.1007/978-3-662-46681-0_4.

[7] Alessandro Cimatti, Chun Tian & Stefano Tonetta (2019): *Assumption-Based Runtime Verification with Partial Observability and Resets*. In: *RV*, *Lecture Notes in Computer Science* 11757, Springer, pp. 165–184, doi:10.1007/978-3-030-32079-9_10.

[8] Alessandro Cimatti, Chun Tian & Stefano Tonetta (2019): *NuRV: A nuXmv Extension for Runtime Verification*. In Bernd Finkbeiner & Leonardo Mariani, editors: *Runtime Verification*, Springer International Publishing, Cham, pp. 382–392, doi:10.1007/978-3-030-32079-9_23.

[9] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez & Vladlen Koltun (2017): *CARLA: An Open Urban Driving Simulator*. In Sergey Levine, Vincent Vanhoucke & Ken Goldberg, editors: *Proceedings of the 1st Annual Conference on Robot Learning*, *Proceedings of Machine Learning Research* 78, PMLR, pp. 1–16, doi:10.48550/arXiv.1711.03938.

[10] Tommaso Dreossi, Daniel J. Fremont, Shromona Ghosh, Edward Kim, Hadi Ravanbakhsh, Marcell Vazquez-Chanlatte & Sanjit A. Seshia (2019): *VerifAI: A Toolkit for the Formal Design and Analysis of Artificial Intelligence-Based Systems*. In Isil Dillig & Serdar Tasiran, editors: *Computer Aided Verification*, Springer International Publishing, Cham, pp. 432–442, doi:10.1007/978-3-030-25540-4_25.

[11] O. foretellix: *Open M-SDL*. `https://releases.asam.net/OpenSCENARIO/2.0-concepts/M-SDL_LRM_OS.pdf`. Accessed: 2023-08-07.

[12] Simone Fratini, Patrick Fleith, Nicola Policella, Alberto Griggio, Stefano Tonetta, Srajan Goyal, Thi Thieu Hoa Le, Jacob Kimblad, Chun Tian, Konstantinos Kapellos, Christos Tranoris & Quirien Wijnands (2023): *Verification and Validation of Autonomous Systems with Embedded AI: The VIVAS Approach*. In: *ASTRA*, p. To appear. Available at `https://az659834.vo.msecnd.net/eventsairwesteuprod/production-atpi-public/070740b67e5b4a32a9be94228c9ac40d`.

[13] Daniel J. Fremont, Edward Kim, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli & Sanjit A. Seshia (2022): *Scenic: a language for scenario specification and data generation*. *Machine Learning*, doi:10.1007/s10994-021-06120-5.

[14] Xiaowei Huang, Marta Kwiatkowska, Sen Wang & Min Wu (2017): *Safety Verification of Deep Neural Networks*. In Rupak Majumdar & Viktor Kunčak, editors: *Computer Aided Verification*, Springer International Publishing, Cham, pp. 3–29, doi:10.1007/978-3-319-63387-9_1.

[15] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian & Mykel J. Kochenderfer (2017): *Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks*. In: *CAV (1)*, *Lecture Notes in Computer Science* 10426, Springer, pp. 97–117, doi:10.1007/978-3-319-63387-9_5.

[16] Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljic, David L. Dill, Mykel J. Kochenderfer & Clark W. Barrett (2019): *The Marabou Framework for Verification and Analysis of Deep Neural Networks*. In: *CAV (1)*, *Lecture Notes in Computer Science* 11561, Springer, pp. 443–452, doi:10.1007/978-3-030-25540-4_26.

[17] Moritz Klischat & Matthias Althoff (2020): *Synthesizing Traffic Scenarios from Formal Specifications for Testing Automated Vehicles*. In: *IV*, IEEE, pp. 2065–2072, doi:10.1109/IV47402.2020.9304617.

[18] Rupak Majumdar, Aman Mathur, Marcus Pirron, Laura Stegner & Damien Zufferey (2021): *Paracosm: A Test Framework for Autonomous Driving Simulations*. In Esther Guerra & Mariëlle Stoelinga, editors: *Fundamental Approaches to Software Engineering*, Springer International Publishing, Cham, pp. 172–195, doi:10.1007/978-3-030-71500-7_9.

[19] Corina S. Păsăreanu, Ravi Mangal, Divya Gopinath, Sinem Getir Yaman, Calum Imrie, Radu Calinescu & Huafeng Yu (2023): *Closed-Loop Analysis of Vision-Based Autonomous Systems: A Case Study*. In Constantin Enea & Akash Lal, editors: *Computer Aided Verification*, Springer Nature Switzerland, Cham, pp. 289–303, doi:10.1007/978-3-031-37706-8_15.

[20] Clément Robert, Jérémie Guiochet, Héléne Waeselynck & Luca Vittorio Sartori (2021): *TAF: a Tool for Diverse and Constrained Test Case Generation*. In: *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pp. 311–321, doi:10.1109/QRS54544.2021.00042.

[21] Luca Vittorio Sartori, Hélène Waeselynck & Jérémie Guiochet (2023): *Pairwise Testing Revisited for Structured Data With Constraints*. In: *ICST*, IEEE, pp. 199–209, doi:10.1109/ICST57152.2023.00027.

[22] Hao Shao, Letian Wang, Ruobing Chen, Hongsheng Li & Yu Liu (2023): *Safety-Enhanced Autonomous Driving Using Interpretable Sensor Fusion Transformer*. In Karen Liu, Dana Kulic & Jeff Ichnowski, editors: *Proceedings of The 6th Conference on Robot Learning*, *Proceedings of Machine Learning Research* 205, PMLR, pp. 726–737, doi:10.48550/arXiv.2207.14024.

[23] Gagandeep Singh, Timon Gehr, Markus Püschel & Martin Vechev (2019): *An Abstract Domain for Certifying Neural Networks*. *Proc. ACM Program. Lang.* 3(POPL), doi:10.1145/3290354.

[24] CARLA Team: *CARLA Autonomous Driving Leaderboard*. `https://leaderboard.carla.org/leaderboard/`. Accessed: 2023-08-30.

[25] CARLA Team: *CARLA ScenarioRunner*. `https://carla-scenariorunner.readthedocs.io`. Accessed: 2023-08-30.

[26] Cumhur Erkan Tuncali, Georgios Fainekos, Hisahiro Ito & James Kapinski (2018): *Sim-ATAV: Simulation-Based Adversarial Testing Framework for Autonomous Vehicles*. In: *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (Part of CPS Week)*, HSCC '18, Association for Computing Machinery, New York, NY, USA, p. 283–284, doi:10.1145/3178126.3187004.

[27] Eric Vin, Shun Kashiwa, Matthew Rhea, Daniel J. Fremont, Edward Kim, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli & Sanjit A. Seshia (2023): *3D Environment Modeling for Falsification and Beyond with Scenic 3.0*. In Constantin Enea & Akash Lal, editors: *Computer Aided Verification*, Springer Nature Switzerland, Cham, pp. 253–265, doi:10.1007/978-3-031-37706-8_13.

[28] Hermann Winner, Karsten Lemmer, Thomas Form & Jens Mazzega (2019): *PEGASUS—First Steps for the Safe Introduction of Automated Driving*. In Gereon Meyer & Sven Beiker, editors: *Road Vehicle Automation 5*, Springer International Publishing, Cham, pp. 185–195, doi:10.1007/978-3-319-94896-6_16.

[29] Penghao Wu, Xiaosong Jia, Li Chen, Junchi Yan, Hongyang Li & Yu Qiao (2022): *Trajectory-guided Control Prediction for End-to-end Autonomous Driving: A Simple yet Strong Baseline*. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho & A. Oh, editors: *Advances in Neural Information Processing Systems*, 35, Curran Associates, Inc., pp. 6119–6132, doi:10.48550/arXiv.2206.08129.